Algorithmen und Datenstrukturen

Foliensatz, 25.11.05

Wintersemester 2005/2006

Strukturierungskonzepte

Strukturierungskonzepte

- Richtige Algorithmen
  - Richtige Programme
    - Definitionen zur (qualitativen) Bewertung von Algorithmen
      - Wert heiße zulässig, wenn Wert innerhalb größten verfügbaren Wertebereichs
        (Ganze Zahlen in 3AA oder Java liegen im Bereich von -2^{63} ... $+2^{63}-1$)
      - Eingabewerte heiße zulässig, wenn zugehöriger Ausgabewert zulässig
    - Programm heißt funktional richtig, wenn es für sämtliche zulässige Eingabewerte die korrekten Ausgabewerte berechnet
    - funktional richtiges Programm heißt richtig, wenn es Ergebnis in einer problemadäquaten (angemessenen) Zeit bestimmt
  - Programm muss
    - alle darstellbaren Werte richtig berechnen
    - Ergebnis in einer angemessen Zeit abliefern
      - 'angemessen' ist eine Zeit in der Größenordnung der Zeit des schnellsten Algorithmus, der diese Funktionalität realisiert.

Algorithmen und Maschinen

- Richtige Algorithmen
  - Beispiel: Binomialkoeffizient
    - Implementiere Formel

$$\binom{m}{n} = \frac{m!}{n! \cdot (m-n)!}.$$

      - möglicher Wertebereichsüberlauf, also falsch!
    - Iteriere in der Reihenfolge (wobei $n = \min(n, m-n)$)

$$\binom{m}{n} = m \cdot (m-1) \cdot \frac{1}{2} \cdot (m-2) \cdot \frac{1}{3} \cdot \ldots \cdot (m-n+1) \cdot \frac{1}{n}.$$

      - möglicher Wertebereichsüberlauf, also falsch!
    - Iteriere in der Reihenfolge (wobei $n = \min(n, m-n)$)

$$\binom{m}{n} = m \cdot \frac{1}{2} \cdot (m-1) \cdot \frac{1}{3} \cdot (m-2) \cdot \ldots \cdot \frac{1}{n} \cdot (m-n+1).$$

      - möglicher Rundungsfehler, also falsch!

Algorithmen und Maschinen

- Richtige Algorithmen
  - Beispiel: Binomialkoeffizient
    - Iteriere in der Reihenfolge
      ($n = \min(n, m-n)$, $g_n^m = \mathrm{ggt}(m,n)$, $r_n^m = n/g_n^m$)

$$\binom{m}{n} = m \cdot \frac{1}{r_2^{m-1}} \cdot \frac{m-1}{g_2^{m-1}} \cdot \frac{1}{r_3^{m-2}} \cdot \frac{m-2}{g_3^{m-2}} \cdot \frac{1}{r_4^{m-3}} \cdot \frac{m-3}{g_4^{m-3}} \cdot \ldots \cdot \frac{1}{r_{m-n+1}^{m-1}} \cdot \frac{m-n+1}{g_{m-n+1}^{m-1}}.$$

      - Richtig, da zunächst mit Teiler gekürzt und dann mit Faktor erweitert wird und die Funktion monoton wächst

Strukturierungskonzepte

- Strukturierungskonzepte in Programmiersprachen
  - mehrere hundert Programmiersprachen
    - viele vermutlich nur von ihren Erfindern benutzt
    - Ideen, die für spezielle Anwendungen
    - für allgemeine Zwecke jedoch wenig nützlich
  - Zweck höherer Programmiersprachen
    - Strukturierung von Information
  - Programmiersprache
    - Basisbefehle
      - Änderung von Werten
      - sehr viele Befehle
      - in einer bestimmten Reihenfolge auszuführen
      - alle nötigen Befehle kennen
      - Ausführungsreihenfolge angeben
    - Zu theoretische Sichtweise

Strukturierungskonzepte

- Zu große Anzahl von Befehlen
- tatsächliche Wirkungsweise unverstehbar
  - Mensch besitzt nur beschränkte Übersicht über komplexe Strukturen
  - Angabe einer zu großen Anzahl von Einzelschritten undurchdringbar
    - Programmierer
    - Leser eines solchen Programms

Strukturierungskonzepte

- 1. Schritt zur Verringerung der Komplexität
  - Verringerung der Anzahl der Schritte, die ein Programm ausführt
    - Programm wird kompakter
      - gleiche Funktionalität
      - Zahl der Maschinenbefehle kann nicht kleiner werden
      - nur die Anzahl der Befehle in höherer Programmiersprache kleiner
    - Beispiel: Auswertung komplexer arithmetischer Ausdrücke
      - `Hilf1 := PreisTopf + PreisDeckel;`
      - `Hilf2 := Hilf1 * Anzahl;`
      - `Betrag := Hilf2 * 1.16 -- Mehrwertsteuer`
    - Rechnung nur in mehreren Schritten nachvollziehbar und überprüfbar
    - Alternative
      - `Betrag := (PreisTopf + PreisDeckel) * Anzahl * 1.16`
    - Überprüfung sehr viel leichter

Strukturierungskonzepte

- 1. Schritt zur Verringerung der Komplexität (ff)
  - kompaktere mathematische Notation verlangt
    - Programmiersprache kennt Konzept
    - der arithmetischen Operation
      - Addition, Subtraktion, ...
    - des arithmetischen Ausdrucks
      - Klammerung, Vorrangregeln, ...
  - verschiedene Grade der Abstraktion von Programmen
    - erste höhere Programmiersprachen wertete mathematische Ausdrücke aus
      - FORTRAN (FORmula TRANslator = Formelübersetzer)

Strukturierungskonzepte

- 2. Schritt zur Verringerung der Komplexität
  - komplexere Probleme, umfangreiche Anweisungsfolgen
    - nicht mehr arithmetisch oder allgemein mathematisch darstellbar
    - Befehlsfolgen durch viele Sprünge (goto) organisiert
    - stark verzweigte, unübersichtliche Programmen
      - unverständlich
    - statt beliebiger Sprünge möglichst nur noch Verzweigungen/Schleifenstrukturen
      - gleiche Befehlsfolgen wie mit goto möglich
      - Leser eines Programms erkennt Fälle einfacher
      - *strukturiertes Programmieren*
      - 1960 in Algol
      - Algol68
      - Pascal von N. Wirth

Strukturierungskonzepte

- 3. Schritt zur Verringerung der Komplexität
  - Verwendung von Prozeduren oder Funktionen
    - zunächst: gleiche Befehlsfolgen zusammenzufassen
      - Speicherplatz sparen
      - Befehlsfolgen haben eigene Bedeutung
        - „Löse ein Gleichungssystem“
        - „Ziehe die Wurzel“
        - Erweiterung der Funktionalität einer Programmiersprache
        - Reduktion des eigentlichen Programmtexts zur Folge
        - höhere Abstraktion

Strukturierungskonzepte

- 4. Schritt zur Verringerung der Komplexität
  - Daten strukturieren
    - Datensätzen (record oder structure)
    - seit langem in kommerzieller Datenverarbeitung angewendet worde
      - COBOL (COmmon Business Oriented Language)
      - noch heute sehr verbreitete Sprache
      - wissenschaftliche Informatik erkannte diese Problematik erst sehr viel später
      - erst in Algol68 und Pascal ähnliche Konstrukte

Strukturierungskonzepte

- 5. Schritt zur Verringerung der Komplexität
  - möglichen Operationen auf Daten direkt zuordnen
    - Zahlen natürlicherweise arithmetischen Operationen
    - Listen: Operationen wie Ein- und Ausketten, Vertauschen, Suchen, Sortieren usw.
  - *Modul (module)*
    - Modell eines realen Systems
      - Menge von Merkmalen
      - bestimmten Änderungen dieser Merkmale
      - Merkmale nur noch von bestimmten Operatoren veränderbar
      - Merkmale vor beliebiger Änderung schützen
        - nicht mehr von jeder Stelle aus beliebig auf bestimmte Werte zugreifbar
      - Programmiersprache
        - MODULA 2 von N. Wirth
        - Ada

Strukturierungskonzepte

- 6. Schritt zur Verringerung der Komplexität
  - möglichen Operationen auf Daten direkt zuordnen
    - Zahlen natürlicherweise arithmetischen Operationen
    - Listen: Operationen wie Ein- und Ausketten, Vertauschen, Suchen, Sortieren usw.
  - *Modul (module)*
    - Modell eines realen Systems
      - Menge von Merkmalen
      - bestimmten Änderungen dieser Merkmale
      - Merkmale nur noch von bestimmten Operatoren veränderbar
      - Merkmale vor beliebiger Änderung schützen
        - nicht mehr von jeder Stelle aus beliebig auf bestimmte Werte zugreifbar
      - Programmiersprache
        - MODULA 2 von N. Wirth
        - Ada

Strukturierungskonzepte

- 6. Schritt zur Verringerung der Komplexität (ff)
  - Nachteile eines Moduls
    - eigentlich nur ein System mit einem Modul beschreibbar
    - verschiedene Instanzen des Moduls durch Kopieren des Programmtexts
      - unbefriedigend
        - zukünftige Änderungen des Moduls in Kopien unberücksichtigt
          - Korrekturen
          - Erweiterungen
          - Löschungen

Strukturierungskonzepte

- 7. Schritt zur Verringerung der Komplexität
  - Eigenschaften auch in anderen Modulen verwenden
    - *Vererbung*
    - **Klasse** (*class*)
      - Beschreibung der Eigenschaften eines Moduls
      - Typ eines Moduls, von Klassifikation kürzer ***class***
      - Typbeschreibungen anderer Klassen ererben
      - Datenzugriff schützen
      - verschiedene Instanzen gleicher Typbeschreibungen erzeugen
    - Zusammenfassung der bisher vorgestellten Konzepte
      - Klassen 1967 in Programmiersprache *SIMULA67* eingeführt
      - erst durch Programmiersprache *Smalltalk* aus den achtziger Jahren populär
      - verbreitetsten Programmiersprachen, heute
        - C++,
        - Java
        - Objective-C
        - Eiffel

Strukturierungskonzepte

- 8. Schritt zur Verringerung der Komplexität
  - Instanzen von Klassen
    - Modelle von Systemen
    - Werte dieser Instanzen ändern sich durch ein kontrollierendes Programm
  - Modellierungskonzept
    - jede Instanz ist passives Objekt
    - durch kontrollierende Instanz (Supervisor) überwacht
    - wenig realistisch
      - reale Welt besteht in der Regel aus verschiedenen aktiven Instanzen
      - Merkmale ändern sich mit der Zeit selbsttätig
      - wirken auf auf Umgebung ein
      - werden von der Umgebungen beeinflusst
      - begrenzte Lebenszeit haben
        - Lebenszeit des Programms länger

Strukturierungskonzepte

- 8. Schritt zur Verringerung der Komplexität
  - sämtlichen Instanzen aktives Eigenleben verschaffen
    - mit anderen Instanzen kommunizieren
    - andere Instanzen dynamisch erzeugen
    - eigenes Leben beliebig beenden
  - ACTOR-Konzept [Hewitt77], in sechziger Jahren entwickelt
    - über experimentelle Ansätze nicht hinaus gekommen
  - Java
    - Konzept des parallelen Programms,
      - Thread
    - eigenständige Prozesse
    - verwalten lokale Variable
    - können gemeinsam mit anderen Threads auf gewisse Werte zugreifen
    - Kommunizieren
    - Synchronisieren

Strukturierungskonzepte

- Motivation der Konzepte
  - meisten Konzepte aus Frühzeit der Informatik
    - Effizienzgründen
    - Unverständnis
    - SIMULA67
    - umfasst die meisten Konzepte
      - paralleler Prozesse
      - ACTOR-Konzept

Syntax von Ausdrücken

- Beschreibung der Syntax von Ausdrücken
  - Darstellung von Ausdrücken oft informal
    - verbale Erklärung über Form eines Ausdrucks
    - in der Informatik jedoch nicht ausreichend
      - Form bestimmt alleine über die Bedeutung
      - anhand der Form kann Maschine (ein Compiler) entscheiden, ob Ausdruck richtig
    - Beschreiben der Form
      - zulässige Form eines beliebigen informatischen Ausdrucks
        - hohe Flexibilität
        - Allgemeinheit
        - Verständlichkeit
        - Einfachheit
      - Darstellung eines Ausdrucks wird als dessen ***Syntax*** bezeichnet
      - Bedeutung eines Ausdrucks wird als dessen ***Semantik*** bezeichnet

Syntax von Ausdrücken

- Grammatik (grammar) beschreibt Syntax
  - System von Regeln
  - sukzessiv anzuwenden
  - schrittweise Ableitung eines Ausdrucks
    - `Ausdruck → Ausdruck '+' Ausdruck  1)`
    - `Ausdruck → Ausdruck  '-' Ausdruck  2)`
    - `Ausdruck → Ausdruck  '*' Ausdruck  3)`
    - `Ausdruck → Ausdruck  '/' Ausdruck  4)`
    - `Ausdruck → '(' Ausdruck ')'  5)`
    - `Ausdruck → Variable  6)`
    - `Ausdruck → Zahl  7)`
  - Ableitung eines Ausdrucks
    - `Zahl`
    - `(Zahl)`
    - `(Variable + Zahl)`
    - `Variable * Zahl + Zahl`

Syntax von Ausdrücken

- Grammatik
  - Ableitung
    - `Ausdruck → Ausdruck '+' Ausdruck`
      `→ Ausdruck '*' Ausdruck '+' Ausdruck`
      `→ Ausdruck '*' Ausdruck '+' Zahl`
      `→ Ausdruck '*' Zahl '+' Zahl`
      `→ Variable '*' Zahl '+' Zahl`
    - `Variable * Zahl + Zahl`
  - Nichtterminalsymbole
    - `Ausdruck`
  - Terminalsymbole
    - `'+', '(', .. Variable, Zahl`
  - Metasymbole
    - `→`

Syntax von Ausdrücken

- Grammatik
  - Satzform
    - abgeleitete Zeichenfolge
  - Wort
    - Satzform nur aus Terminalsymbolen
  - Grammatik
    - alle Wörter, die eine solche Grammatik erzeugt
  - Eine Grammatik ist ein Tupel G = (N, T, P, S),
    - N = Menge der Nichtterminalsymbole
    - T = Menge der Terminalsymbole
    - P = Menge der Produktionen oder Regeln
    - S = Ein Startsymbol

Syntax von Ausdrücken

- Grammatik
  - Verschiedene Sprachklassen durch Struktur der Grammatik
    - Chomsky-Hierarchie
      - Theoretische Informatik
    - Hier nur zwei Sprachklasse
      - Reguläre Sprache
      - Kontextfreie Sprachen

Syntax von Ausdrücken

- Grammatik für Reguläre Sprachen
  - Regulären Sprachen von endlichen Automaten erzeugt
  - Reguläre Ausdrücke
    - {}, {ε} und {a} mit a ∈ A sind Mengen regulärer Ausdrücke über Alphabet A.
    - Sind zwei Mengen R und S regulärer Ausdrücke gegeben, so beschreibt
      - R* die Menge, die durch Konkatenation (Hintereinanderschreibung) beliebig vieler (auch keiner) Ausdrücke aus R entsteht,
      - RS die Menge, die durch Konkatenation eines regulären Ausdrucks aus R mit einem aus S entsteht,
      - R | S die Menge, die durch Auswahl eines Elements aus R oder S entsteht.
      - Mengen R* , RS und R | S sind Mengen regulärer Ausdrücke über Alphabet A, wenn R und S Mengen regulärer Ausdrücke über dem Alphabet A sind.

Syntax von Ausdrücken

- Grammatik für Reguläre Sprachen
  - Beispiele zu
    - {}, {ε} und {a} mit a ∈ A sind Teilmengen einer regulären Sprache ***S*** regulärer Ausdrücke über Alphabet A
    - sei A={a,b,c,d}
  - {} heißt: jede leere Menge ist Teilmenge jeder regulären Sprache ***S***
  - {ε} heißt: das leere Wort ist auch in jeder regulären Sprache enthalten
    - dies dient nur der rekursiven Definition,
      man braucht sich keine Gedanken über eine 'leeres' Wort zu machen
  - {a} heißt: das Wort 'a' ist ein Wort aus der regulären Sprachen ***S***
  - {b} heißt: das Wort 'b' ist ein Wort aus der regulären Sprachen ***S***
  - {c} heißt: das Wort 'c' ist ein Wort aus der regulären Sprachen ***S***
  - {d} heißt: das Wort 'd' ist ein Wort aus der regulären Sprachen ***S***

Syntax von Ausdrücken

- Grammatik für Reguläre Sprachen
  - Beispiele zu
    - R* die Menge, die durch Konkatenation (Hintereinanderschreibung) beliebig vieler (auch keiner) Ausdrücke aus R = {a, b, c, d} entsteht,
    - $a^0 = \varepsilon \in \boldsymbol{S}$
    - $a^1 = a \in \boldsymbol{S}$
    - $a^{22} = aaaaaaaaaaaaaaaaaaaaaa \in \boldsymbol{S}$
    - $a^5b^3d^1 = aaaaabbbd \in \boldsymbol{S}$
    - $ab^3d = abbbd \in \boldsymbol{S}$

Syntax von Ausdrücken

- Grammatik für Reguläre Sprachen
  - Beispiele zu
    - RS die Menge, die durch Konkatenation eines regulären Ausdrucks aus $R = \{v_1, v_2, v_3, ...\}$ mit einem aus $S = \{w_1, w_2, w_3, ...\}$ entsteht
    - $v_1w_1 \in \boldsymbol{S}$
    - $v_2w_1 \in \boldsymbol{S}$
    - $v_1w_2 \in \boldsymbol{S}$
    - $v_2w_2 \in \boldsymbol{S}$
    - $v_{12321}w_{242} \in \boldsymbol{S}$

Syntax von Ausdrücken

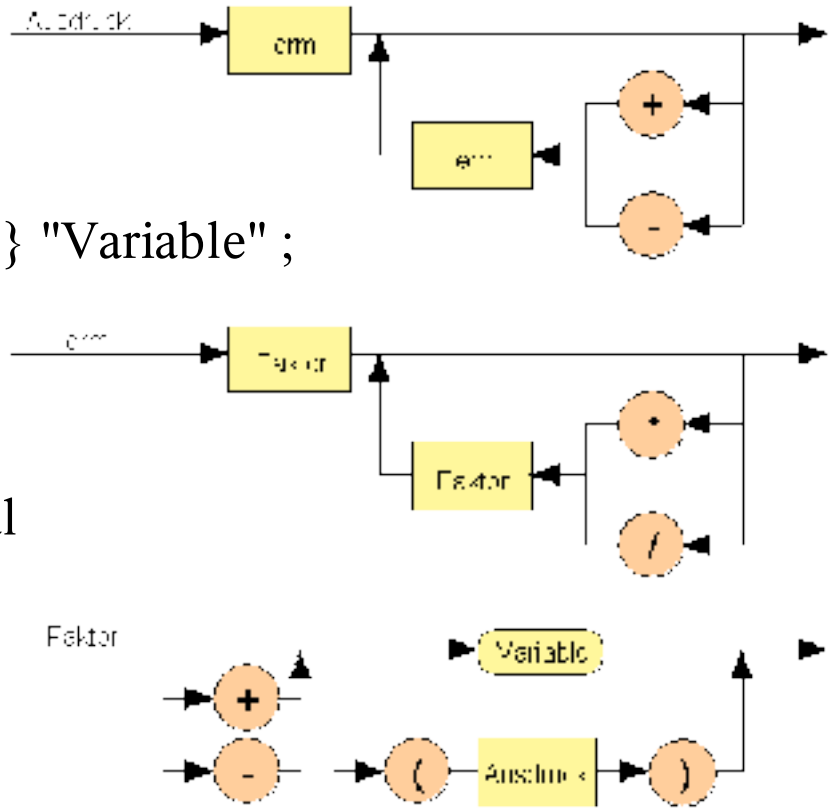
- Grammatik für Reguläre Sprachen
  - Beispiele zu
    - R | S die Menge, die durch Auswahl eines Elements aus R = {v_1, v_2, v_3, ...} oder S = {w_1, w_2, w_3, ...} entsteht
    - $v_1 \in \boldsymbol{S}$
    - $w_1 \in \boldsymbol{S}$
    - $w_2 \in \boldsymbol{S}$
    - $w_{22} \in \boldsymbol{S}$
    - $v_{1111} \in \boldsymbol{S}$

Syntax von Ausdrücken

- Grammatik für Reguläre Sprachen
  - Beispiele
    - $A = G \cup K \cup Z$,
    - G = {A, B, C, .., Z},
    - K = {a, b, c, .., z},
    - Z = {1, 2, ..,9}.
  - ZZ* = {1, 22, 321, 1234543231, ... }
  - GGKKZ* = {ABee, ABee2, ABfx2345, ... }
  - GZK* = {A1, X29abc, W75light, ... }
  - 'Vers.'ZZ*'.'ZZ* = {Vers.1.0, Vers.3.2, Vers.24.6, ... }

Syntax von Ausdrücken

- Grammatik für kontextfreie Sprachen
  - Backus-Naur-Form
    - Ausdruck: Term (("+"|"–") Term)* ;
    - Term: Faktor (("*"|"/") Faktor)* ;
    - Faktor: {"–"} "(" Ausdruck ")" | {"–"} "Variable" ;
  - Hier bedeuten
    - (…|…) genau eine Alternative
    - {…} Inhalt der Klammer ist optional
    - (…)* Inhalt der Klammer kann beliebig oft (auch gar nicht) stehen;
    - am Ende einer Regel steht ein ';';
    - Terminalsymbole werden in doppelten Anführungszeichen "Variable" eingeschlossen.



Syntax von Ausdrücken

- Grammatik
  - Reguläre oder Chomsky-3-Grammatik erzeugt Reguläre Sprache
    - Bezeichner (*identifier*)
      - Name, Anfang, Start, Ende, Wert
    - Schlüsselwörter (*keyword*)
      - von der Sprache definierte Wörter: if, then, while usw.
    - Zahlen: 123, 12.345, 1.0e5
    - Scanner
      - lexikalische Analyse
  - Kontextfreie oder Chomsky-2-Grammatik erzeugt kontextfreie Sprache
    - If (value==0) then x:=0; else x:=1;
  - LL(1), LR(k)
    - Spezielle kontextfreie Sprachen zur Syntaxanalyse
    - Parser
      - syntaktische Analyse